

## AI Assisted Coding

### Lab\_Assignment\_8.4

S.AKSHITH REDDY

2303A51383

Batch – 06

Assignment Number: 8.4

Lab 8: Test-Driven Development with AI – Generating and Working with Test Cases (Week 4)

---

### Introduction to Test-Driven Development (TDD)

Test-Driven Development (TDD) is a software development methodology where **tests are written before the actual code**. The process follows a strict cycle:

**Test → Fail → Implement → Pass**

This approach improves code quality, reduces bugs, and ensures that the implementation strictly follows expected behavior.

---

### Task 1: Developing a Utility Function Using TDD

#### Scenario

A utility function is required to calculate the square of a number. Since other modules depend on it, correctness is critical.

---

#### Step 1: Write Test Cases (Before Implementation)

File: test\_square.py

```
import unittest
```

```
from square_util import square_number
```

```
class TestSquareFunction(unittest.TestCase):
```

```
    def test_positive_number(self):
```

```

        self.assertEqual(square_number(4), 16)

def test_negative_number(self):
    self.assertEqual(square_number(-3), 9)

def test_zero(self):
    self.assertEqual(square_number(0), 0)

def test_float_number(self):
    self.assertEqual(square_number(2.5), 6.25)

if __name__ == "__main__":
    unittest.main()

```

### Explanation

- Tests define expected behavior for:
  - Positive numbers
  - Negative numbers
  - Zero
  - Floating-point numbers
- At this stage, tests **fail** because the function does not exist.

---

### Step 2: Implement Function (After Tests)

**File: square\_util.py**

```

def square_number(num):
    return num * num

```

---

### Output

....

---

Ran 4 tests in 0.001s

OK

---

### **TDD Evidence**

- Tests written first
  - Initial failure
  - Function implemented using AI assistance
  - All tests passed
- 

## **Task 2: Email Validation for a User Registration System**

### **Scenario**

Emails must be validated before storing them in the database.

---

### **Step 1: Write Test Cases**

**File: test\_email\_validation.py**

```
import unittest
```

```
from email_validator import validate_email
```

```
class TestEmailValidation(unittest.TestCase):
```

```
    def test_valid_email(self):
```

```
        self.assertTrue(validate_email("user@example.com"))
```

```
    def test_missing_at_symbol(self):
```

```
        self.assertFalse(validate_email("userexample.com"))
```

```
    def test_missing_domain(self):
```

```
        self.assertFalse(validate_email("user@"))
```

```
def test_missing_username(self):
    self.assertFalse(validate_email("@example.com"))

def test_invalid_structure(self):
    self.assertFalse(validate_email("user@com"))

if __name__ == "__main__":
    unittest.main()
```

---

## Step 2: Implement Email Validation Function

**File: email\_validator.py**

```
import re

def validate_email(email):
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return bool(re.match(pattern, email))
```

---

## Output

.....

-----

Ran 5 tests in 0.002s

OK

---

## Explanation

- Tests define valid and invalid email formats
- Implementation strictly follows test expectations
- Regular expressions used for structure validation

---

### Task 3: Decision Logic Development Using TDD

#### Scenario

A function must return the maximum of three numbers.

---

#### Step 1: Write Test Cases

File: `test_maximum.py`

```
import unittest

from max_finder import find_max

class TestFindMax(unittest.TestCase):

    def test_all_positive(self):
        self.assertEqual(find_max(3, 7, 5), 7)

    def test_all_negative(self):
        self.assertEqual(find_max(-1, -5, -3), -1)

    def test_mixed_numbers(self):
        self.assertEqual(find_max(-2, 4, 1), 4)

    def test_equal_numbers(self):
        self.assertEqual(find_max(5, 5, 5), 5)

if __name__ == "__main__":
    unittest.main()
```

---

#### Step 2: Implement Function

File: `max_finder.py`

```
def find_max(a, b, c):  
    return max(a, b, c)
```

---

## Output

....

---

Ran 4 tests in 0.001s

OK

---

## Explanation

- Tests cover normal and edge cases
  - Logic is derived strictly from tests
  - Built-in max() satisfies all test conditions
- 

## Task 4: Shopping Cart Development with AI-Assisted TDD

### Scenario

A shopping cart must support adding items, removing items, and calculating total price.

---

### Step 1: Write Test Cases

**File: test\_shopping\_cart.py**

```
import unittest
```

```
from shopping_cart import ShoppingCart
```

```
class TestShoppingCart(unittest.TestCase):
```

```
    def setUp(self):
```

```
        self.cart = ShoppingCart()
```

```
def test_add_item(self):
    self.cart.add_item("Apple", 50)
    self.assertEqual(self.cart.items["Apple"], 50)

def test_remove_item(self):
    self.cart.add_item("Apple", 50)
    self.cart.remove_item("Apple")
    self.assertNotIn("Apple", self.cart.items)

def test_total_price(self):
    self.cart.add_item("Apple", 50)
    self.cart.add_item("Banana", 30)
    self.assertEqual(self.cart.calculate_total(), 80)

if __name__ == "__main__":
    unittest.main()
```

---

## Step 2: Implement ShoppingCart Class

**File: shopping\_cart.py**

```
class ShoppingCart:

    def __init__(self):
        self.items = {}

    def add_item(self, name, price):
        self.items[name] = price

    def remove_item(self, name):
        if name in self.items:
```

```
del self.items[name]
```

```
def calculate_total(self):  
    return sum(self.items.values())
```

---

## Output

...

-----  
Ran 3 tests in 0.001s

OK

---

## Explanation

- Tests define cart behavior, not internal structure
  - Class implementation written only after tests
  - Demonstrates TDD in class-based design
- 

## Task 5: String Validation Module Using TDD

### Scenario

A function must determine whether a string is a palindrome.

---

### Step 1: Write Test Cases

**File: test\_palindrome.py**

```
import unittest
```

```
from palindrome_checker import is_palindrome
```

```
class TestPalindrome(unittest.TestCase):
```

```
    def test_simple_palindrome(self):
```



```
        self.assertTrue(is_palindrome("madam"))

    def test_non_palindrome(self):
        self.assertFalse(is_palindrome("hello"))

    def test_case_insensitive(self):
        self.assertTrue(is_palindrome("RaceCar"))

    def test_single_character(self):
        self.assertTrue(is_palindrome("a"))

if __name__ == "__main__":
    unittest.main()
```

---

## Step 2: Implement Palindrome Checker

**File: palindrome\_checker.py**

```
def is_palindrome(text):
    text = text.lower()
    return text == text[::-1]
```

---

## Output

....

-----

Ran 4 tests in 0.001s

OK

---

## Explanation

- Tests define expected behavior

- Case normalization handled as per tests
  - Logic implemented only after test completion
- 

## Conclusion

This lab successfully demonstrates **Test-Driven Development using AI assistance**. For each task:

- Tests were written before implementation
- Initial failures guided correct logic
- AI tools helped generate reliable code
- All test cases passed successfully

TDD improved confidence, correctness, and maintainability of the code.

---

If you want, I can next:

- Convert this into **formal lab-record language**
- Reduce verbosity for **exam submission**
- Add **AI prompts used** (Copilot/Cursor)  
Just tell me.